CSCE 314 Programming Languages – Fall 2015
Hyunyoung Lee
Assignment 7
Assigned on Wednesday, November 11, 2015

Notice the electronic submission deadline extension (originally Monday, Nov. 23)
Electronic submission on eCampus due at 9:00 a.m., Wednesday, 11/25, 2015
Please bring signed coversheet to class on Monday, 11/23, 2015
If you do not turn in a signed coversheet your work will not be graded.

"On my honor, as an Aggie, I have neither given nor received any unauthorized aid on any portion of the academic work included in this assignment."

Typed or printed name of student	Section (501 or 502)
Signature of student	UIN

Note 1: This homework set is *individual* homework, not a team-based effort. Discussion of the concept is encouraged, but actual write-up of the solutions must be done individually.

Note 2: Turn in on eCampus one `yourLastName-yourFirstName-a7.zip` or `.tar` file that contains all your Java programs, your answer files for Problem 4, and a readme.txt that you deem necessary for the grader to compile and run your programs.

Note 3: All Java code that you submit must compile without errors using the `javac` command of Java 8 (no IDE please). If your code does not compile, you will likely receive zero points for this assignment.

Note 4: Remember to put the head comment in your files, including your name and acknowledgements of any help received in doing this assignment. Remember the honor code.

You will earn total 150 points.

Problem 1. (40 points) Implement generic interfaces. The following `Node` class can represent a singly-linked list.

```
public final class Node<T> {
    public final T v;
    public Node<T> next;
    public Node (T val, Node<T> link) { v = val; next = link; }
}
```

Task 1. Define a class `NodeIterator<T>` to iterate over the values stored in a linked list of `Node<T>` objects. The constructor of that class should take a `Node<T>` as a parameter, and thus have the header:

```
public NodeIterator (Node<T> n)
```

Your `NodeIterator<T>` class must implement the `java.util.Iterator<T>` interface (see `https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html`).

Task 2. Then make `Node<T>` *iterable*; see `https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html`.

Task 3. Now, if `list` is of type `Node<T>`, you should be able to iterate over `list` using Java's "for each" `for`-loop:

```
for (T e : list) { /* do something with e */ }
```

Implement the class `Test1` that contains three static methods:

(a) `main()` that tests your `NodeIterator<T>` class,
(b) `sum()` that accepts a linked list of type `Node<Integer>` and sums up the values in each node in the linked list, and
(c) `print()` that accepts a linked list of type `Node<Integer>` and prints out the values in each node in the linked list.

Then, in the `main()`, create the `list` of type `Node<Integer>` with five integers (say, 1 to 5), and then invoke the two methods `print()` and `sum()` passing `list` as the argument.

Problem 2. (50 points) Implement a nicer linked list. You may notice in Problem 1 that it is rather inconvenient to build lists with `Node`. Implement another generic class `LinkedList<T>`, in terms of `Node<T>`, that has a nicer interface.

Task 1 Make `LinkedList<T>` *iterable* by implementing the `Iterable<T>` interface.

Task 2 Define the two constructors for the `LinkedList<T>` class:

```
public LinkedList();  // create an empty list
public LinkedList(Iterable<T> iterable);
```

Task 3 Implement these member methods (with their expected meaning):

```
public LinkedList<T> reverse();
public String toString();
```

A call `x.reverse()` should reverse `x`, and return the reversed `x` as the result. An easy way to implement `reverse()` is to reconstruct a new list, and swap that in place of the original. The `toString()` method should print out the list in Haskell list form, for example, a list of integers 1, 2, and 3 should be printed as `[1,2,3]`.

Task 4 Write a class `Test2` of which `main()` tests your `LinkedList` class thoroughly – all of its methods and constructors should be tested. Here's some example code that should compile and run.

```
LinkedList<Integer> empty_list = new LinkedList<Integer>();
LinkedList<Integer> list =
    new LinkedList<Integer>(Arrays.asList(1, 2, 3, 4, 5, 6));

System.out.println(empty_list);
System.out.println(empty_list.reverse());
System.out.println(list);
System.out.println(list.reverse());

int sum = 0;
for (int e : list) { sum += e; }
System.out.println(sum);
```

The output should be:

```
[]
[]
[1,2,3,4,5,6]
[6,5,4,3,2,1]
21
```

To get `Arrays.asList` in scope, import `java.util.Arrays`.

Problem 3. (30 points) Practice wildcards. Below is a `Shop<T>` class that maintains a stock of objects of type `T`. There are two `sell` functions (a customer sells to the store): one for selling a single item of type `T` and another for selling all items in a `List<T>`. There are two `buy` methods (the customer buys from the store): one to buy a single item of type `T` and the other to buy n items at a time, to be added into a `List<T>` collection provided as an argument to the `buy` function. Note that `LinkedList` is qualified so as not to confuse it with the `LinkedList` from Problem 2.

```
import java.util.LinkedList;
import java.util.List;

public class Shop<T> {
   List<T> stock;

   public Shop() { stock = new java.util.LinkedList<T>(); }
   void sell(T item) {
        stock.add(item);
   }
   public T buy() {
        return stock.remove(0);
   }
   void sell(List<T> items) {
```

```
        for (T e : items) {
              stock.add(e);
        }
    }
    void buy(int n, List<T> items) {
        for (T e : stock.subList(0, n)) {
              items.add(e);
        }
        for (int i=0; i<n; ++i) stock.remove(0);
    }
}
```

Modify the class so that you can buy items into and sell items from any `Collection` type, not just `List`. Also, the element types of the collection should not have to match exactly when buying and selling: allow for maximally flexible (but still static) typing using wildcards. Write `Test3` class that contains a `main()` function that tests your implementation. In your tests, use at least three different collection types. Also, define at least three static nested classes in `Test3` expressing an example subclass hierarchy and use those classes in your tests.

Problem 4. (30 points) More Wildcards. Assume that `Integer` derives from `Number`, and `List<T>` and `Set<T>` from `Collection<T>`.

1. Draw the subtype hierarchy of the following types:

   ```
   List<?>
   List<Integer>
   List<Object>
   List<Number>
   List<? extends Number>
   List<? super Number>
   ```

2. Draw the subtype hierarchy of the following types:

   ```
   Set<Integer>
   List<String>
   Object
   Collection<Integer>
   Collection<Object>
   Collection<?>
   ```

Use any drawing program (e.g., PowerPoint or Keynote) you are familiar with, save (export) your file as Problem4.pdf, and include the PDF file in your a7 directory before creating the .zip or .tar file. Hand-drawn-&-scanned/photographed answers will *not* be accepted.